UNIVERSITY OF ALBERTA

Inter-Procedural Analysis

CMPUT 497/500
Foundations of Program Analysis

Karim Ali
@karimhamdanali

Disclaimer

- Modified slides from Eric Bodden (Paderborn) and Uday Khedker (IIT)

Previously

- Points-to
- Aliases
- Must and May analyses
- Incomplete Programs
- Weak vs Strong Updates
- Access Paths
- Distributivity

Inter-Procedural Data-Flow Analysis

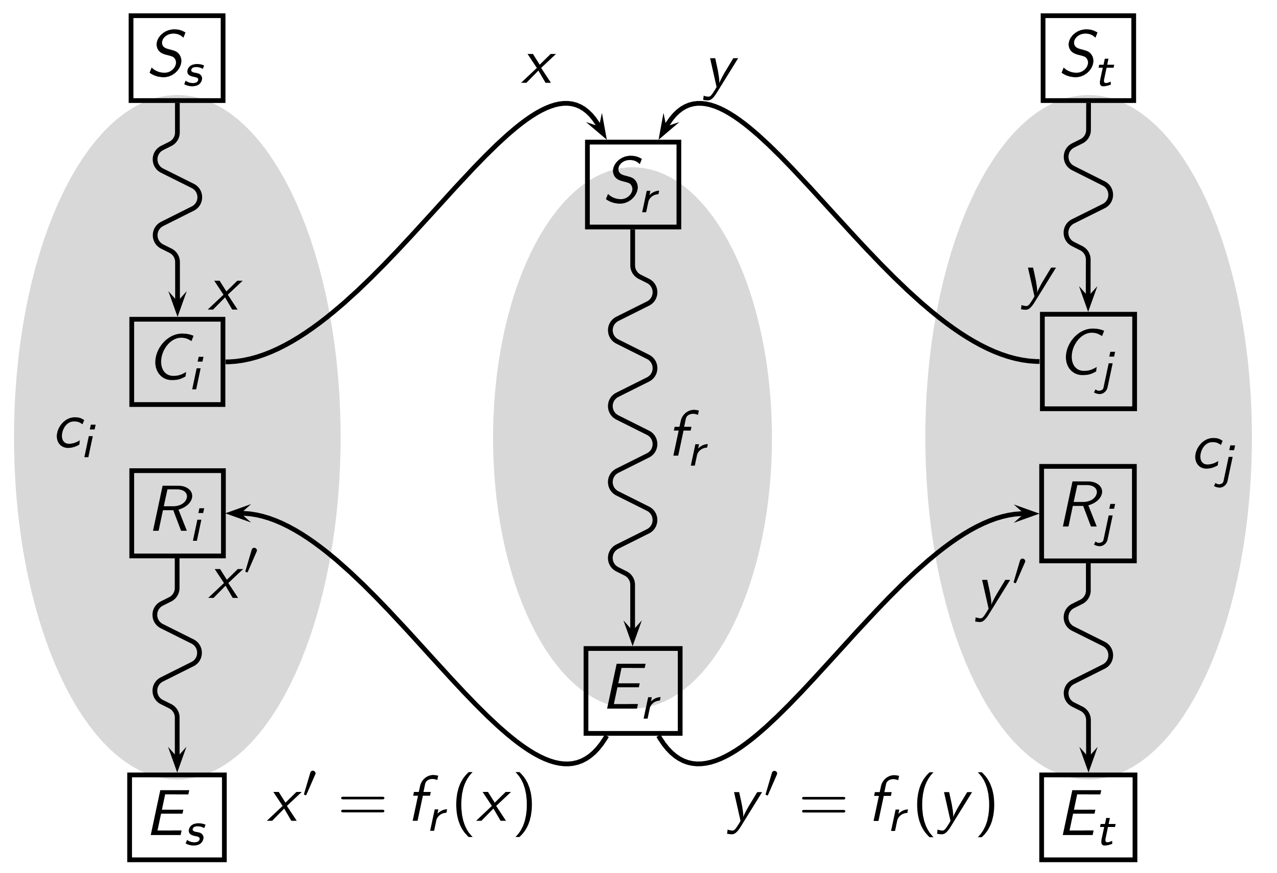
- Beyond procedure boundaries
- Model the effects of
  - calls in the callers, and
  - calling contexts in the callees

Inter-Procedural Data-Flow Analysis

- Approaches
  - Generic: Call-strings approach, functional approach
  - Problem specific: Alias analysis,
    Points-to analysis,
    Partial redundancy elimination,
    Constant propagation

Inter-Procedural Data-Flow Analysis

fun s() fun r() fun t()



Data Flow Information	
x	Inherited by procedure r from call site c_i in procedure s
y	Inherited by procedure r from call site c_j in procedure t
x'	Synthesized by procedure r in s at call site procedure c_i
y'	Synthesized by procedure r in t at call site procedure c_j

Inherited vs Synthesized Analysis Information

Inherited Analysis Information

- Answering questions about formal parameters and global variables:
  - Which variables carry constant values?
  - Which variables aliased with eac other?
  - Which locations can a pointe point to?

May or Must

Synthesized Analysis Information

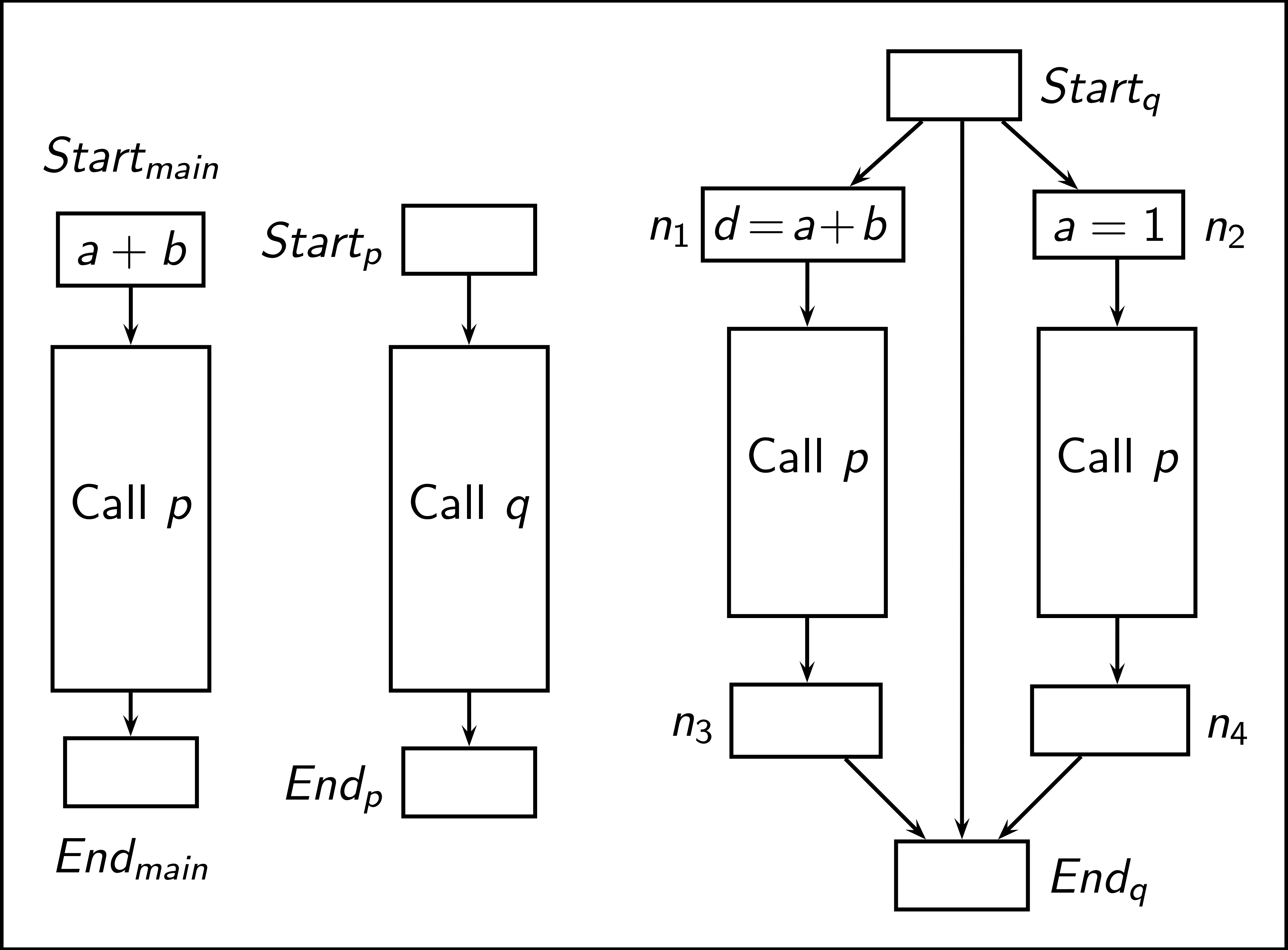
- Answering questions about side-effects of a procedure call:
  - Which local/global/formal variables are defined in a callee?
  - Which local/global/formal variables are used by a callee?

May
or
Must

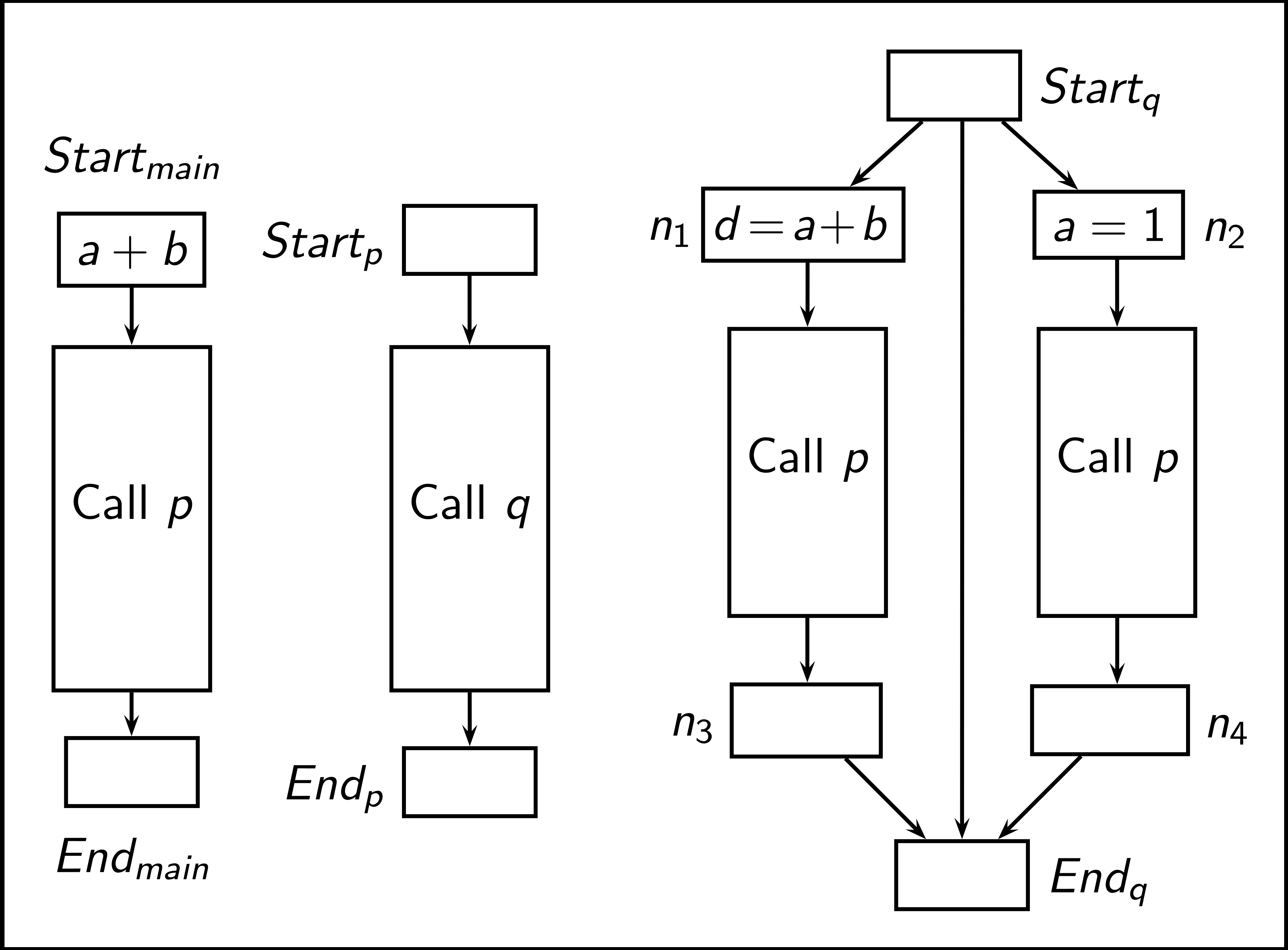
Inter-Procedural Control-Flow Graph (ICFG)

aka “program super-graph”

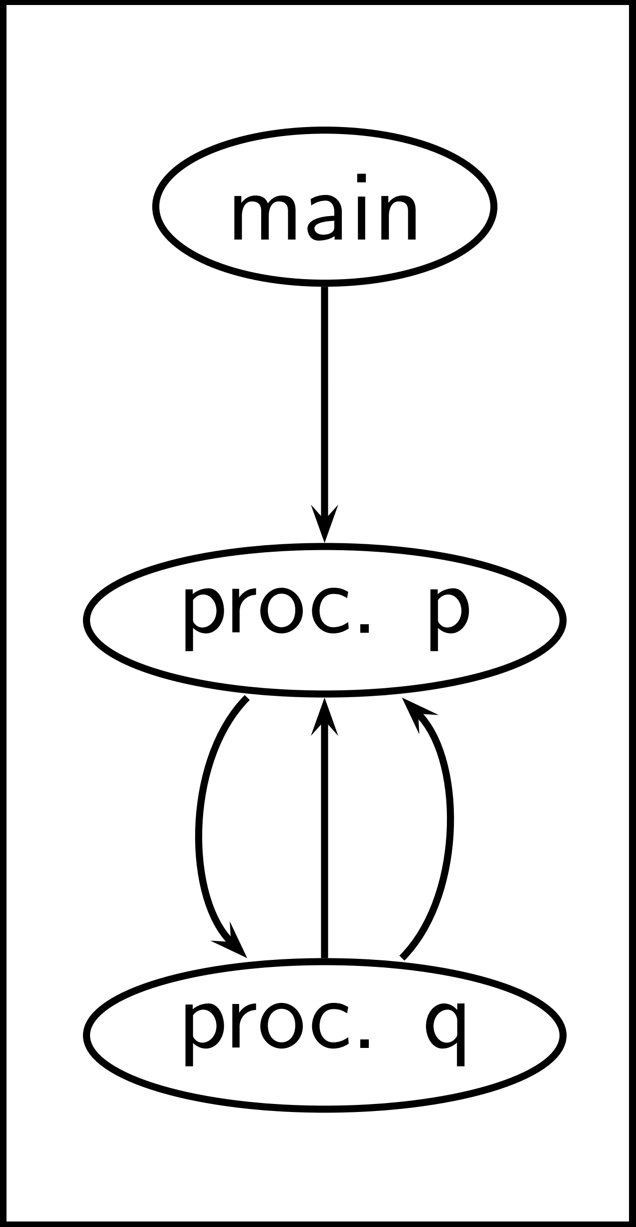
Procedure Space



Call Graph

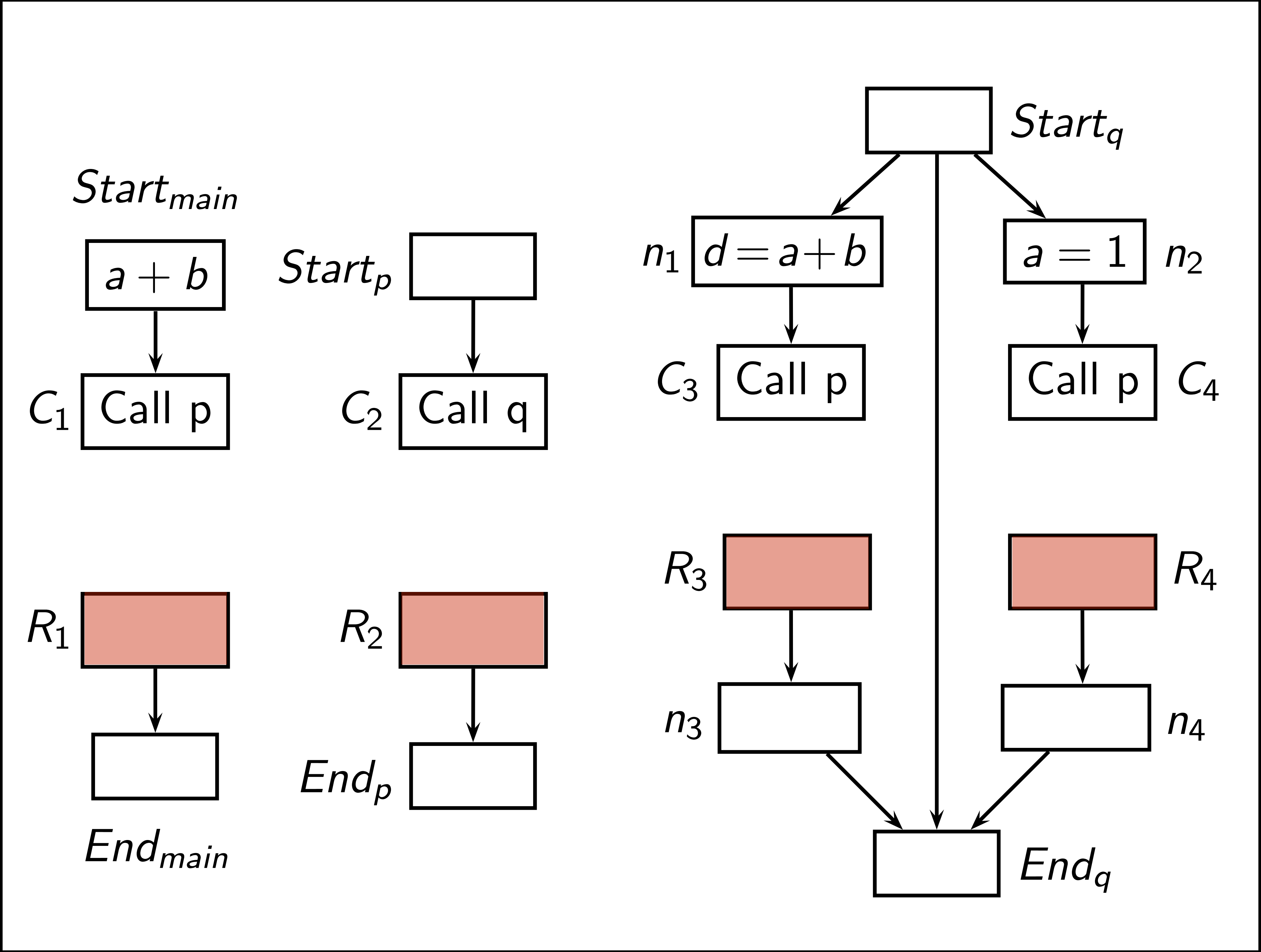


$Start_{main}$
$a + b$
Call p
End_{main}
$Start_p$
Call q
End_p
$Start_q$
n_1
$d = a + b$
$a = 1$
n_2
Call p
Call p
n_3
n_4
End_q

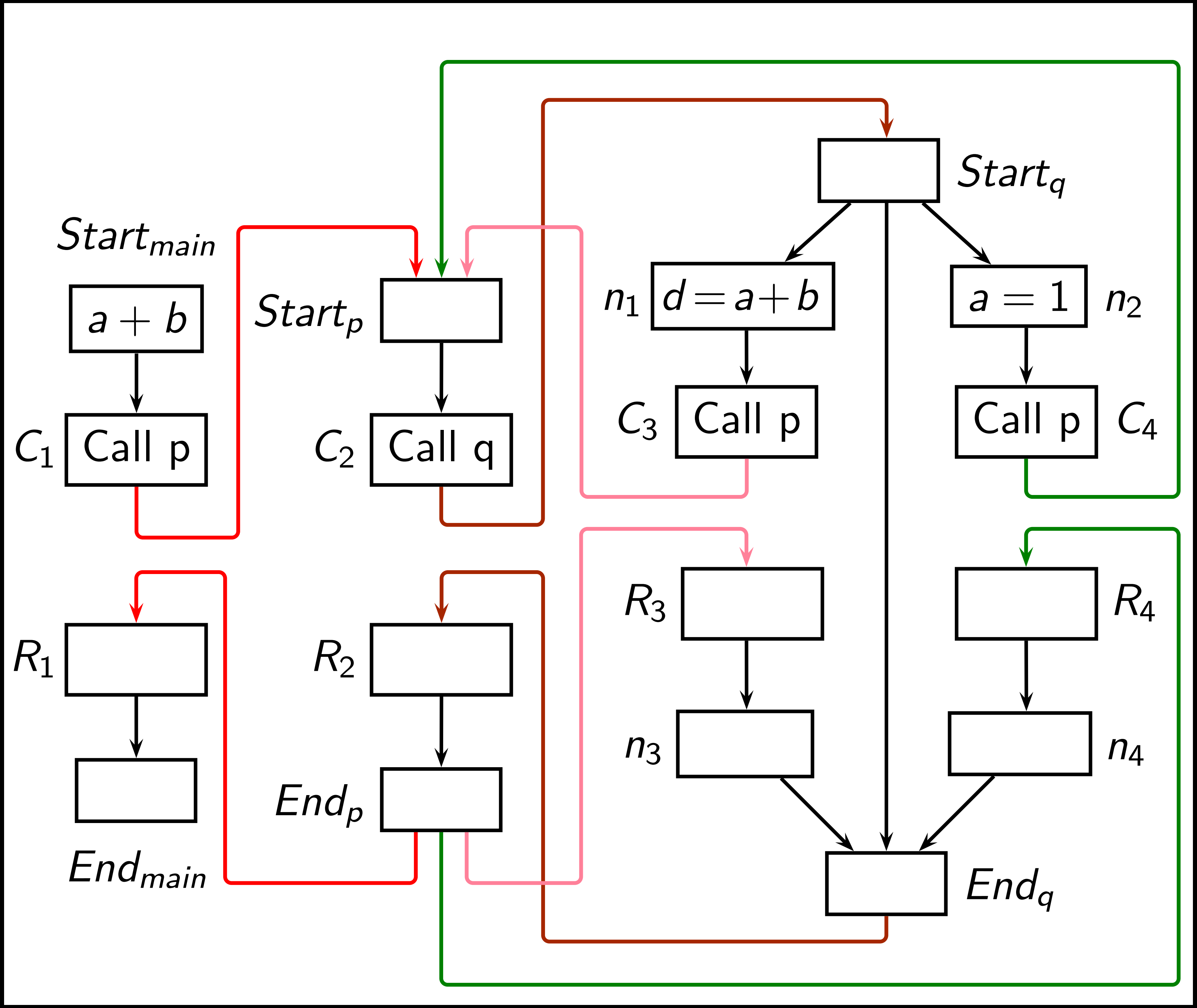


main
proc. p
proc. q

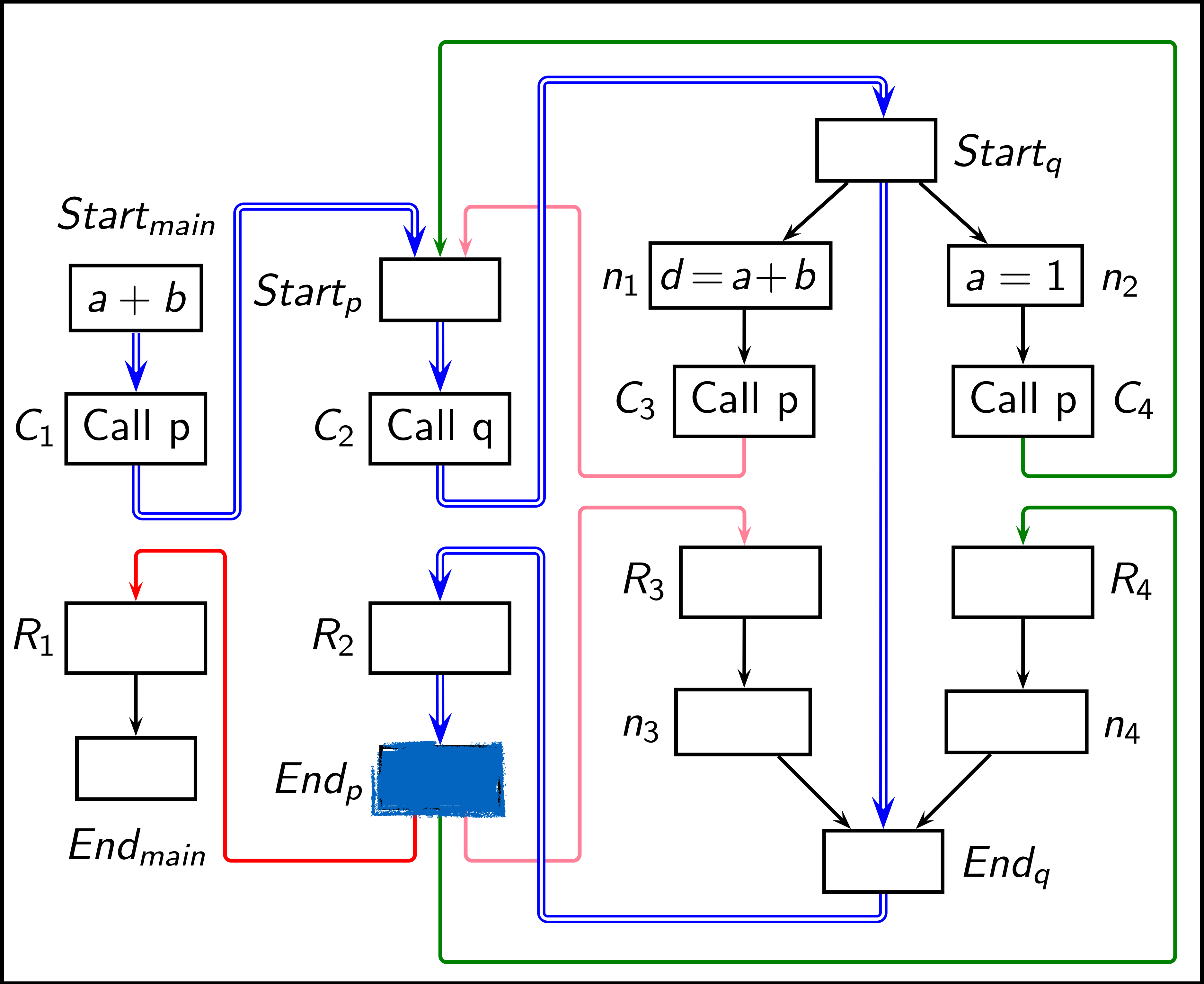
Introducing Return Sites

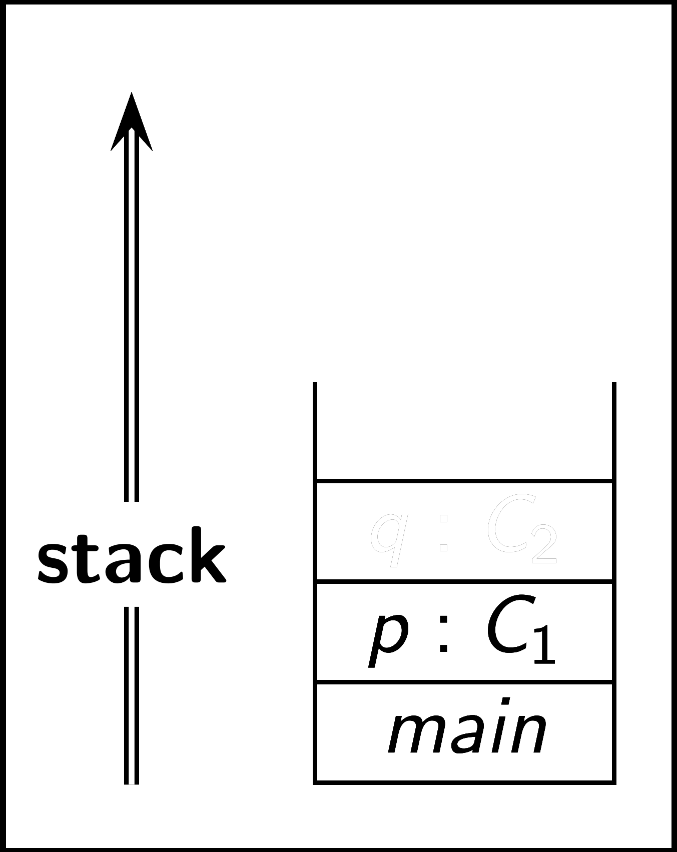


Caller-Callee Relationships

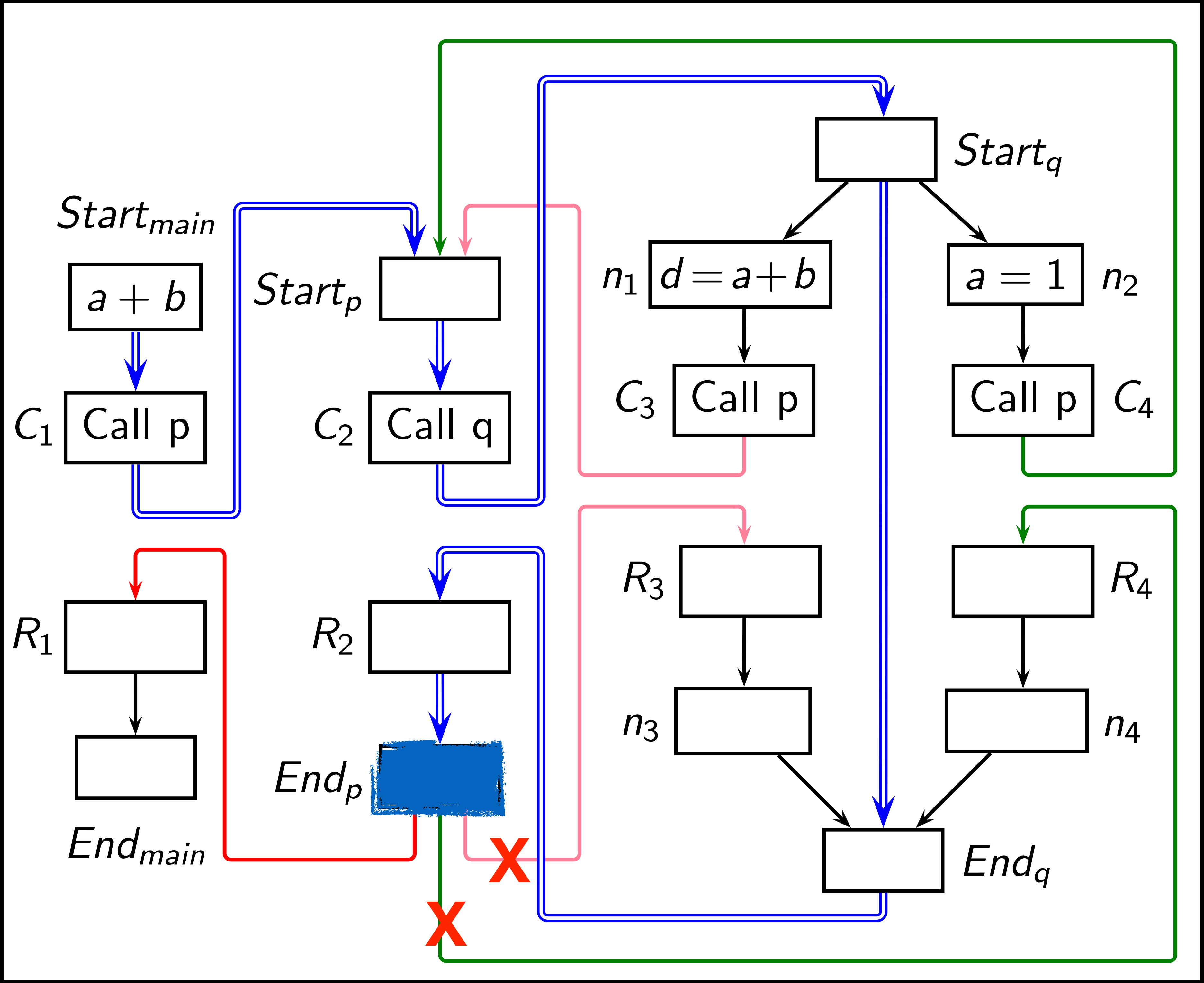


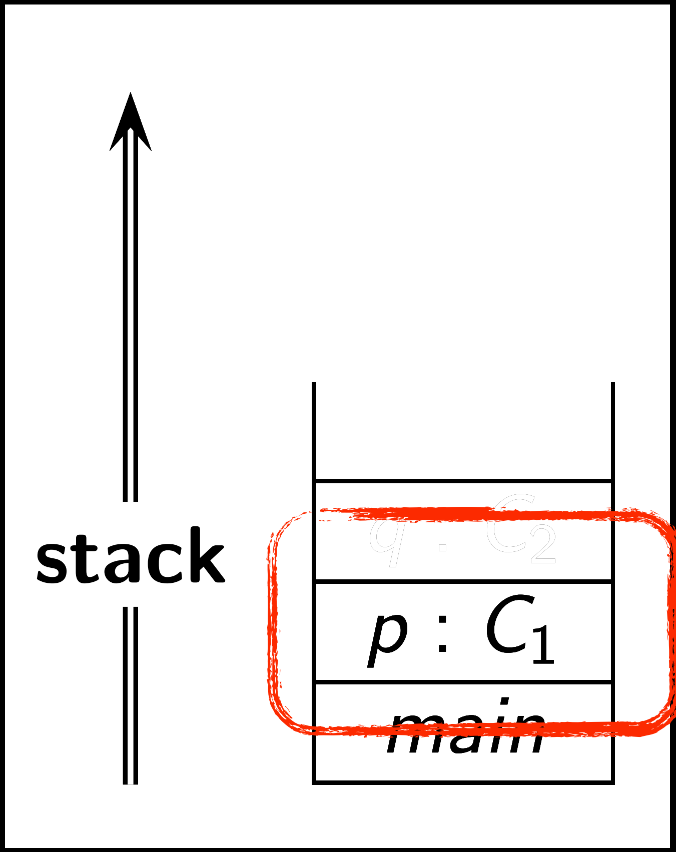
Valid/Realizable Path



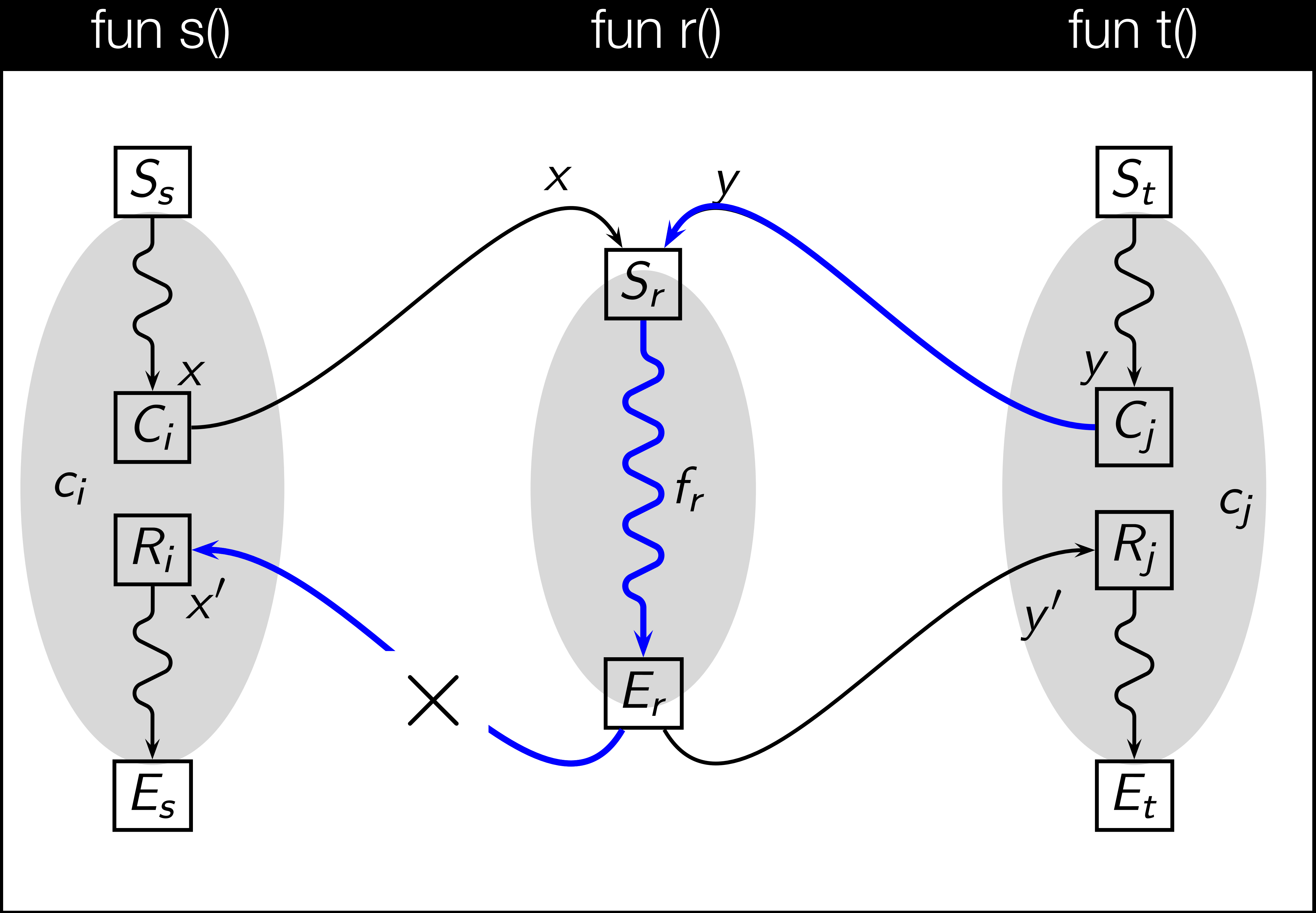


Invalid/Unrealizable Path

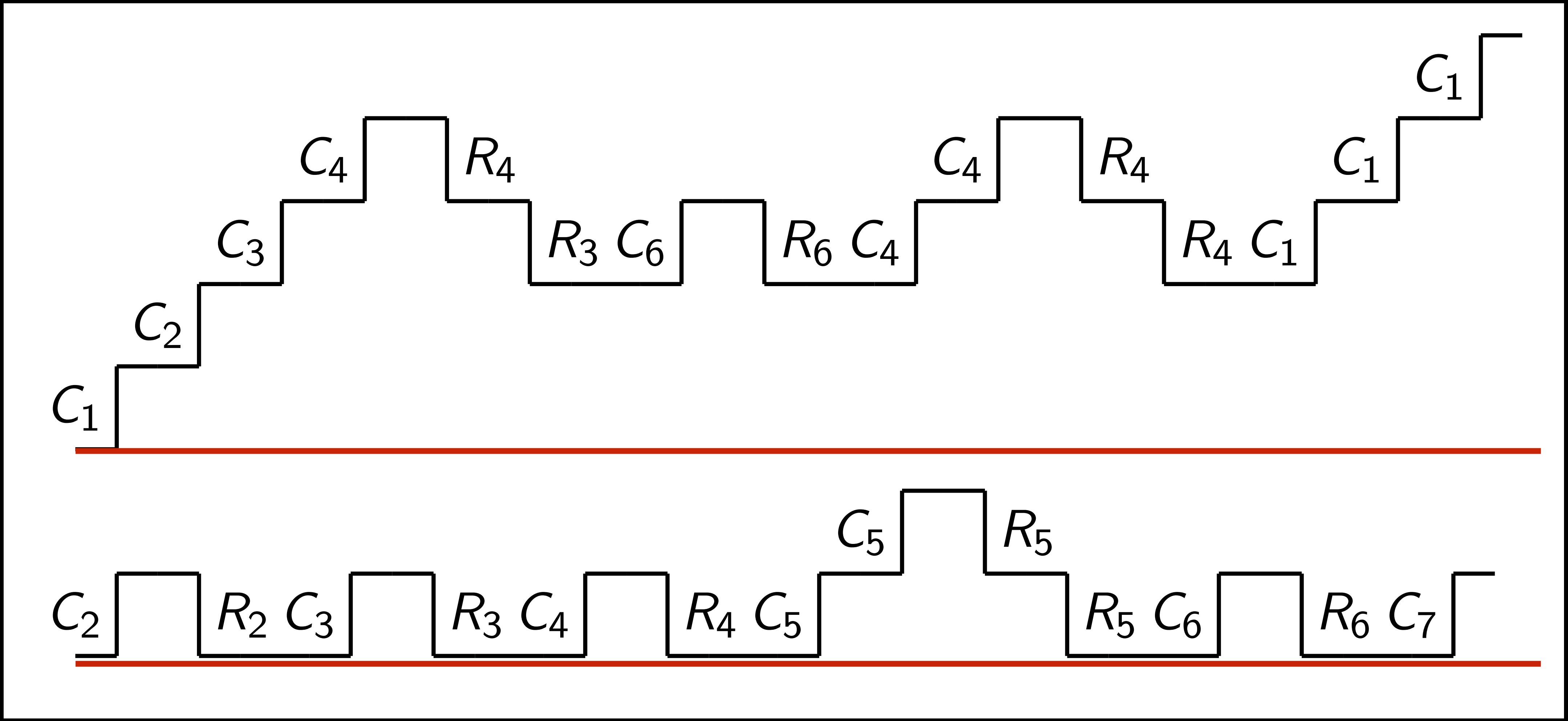




Recognizing Invalid Paths



Staircase of Calls and Returns



You can descend only as much as you have ascended!

Every descending step must match a corresponding ascending step

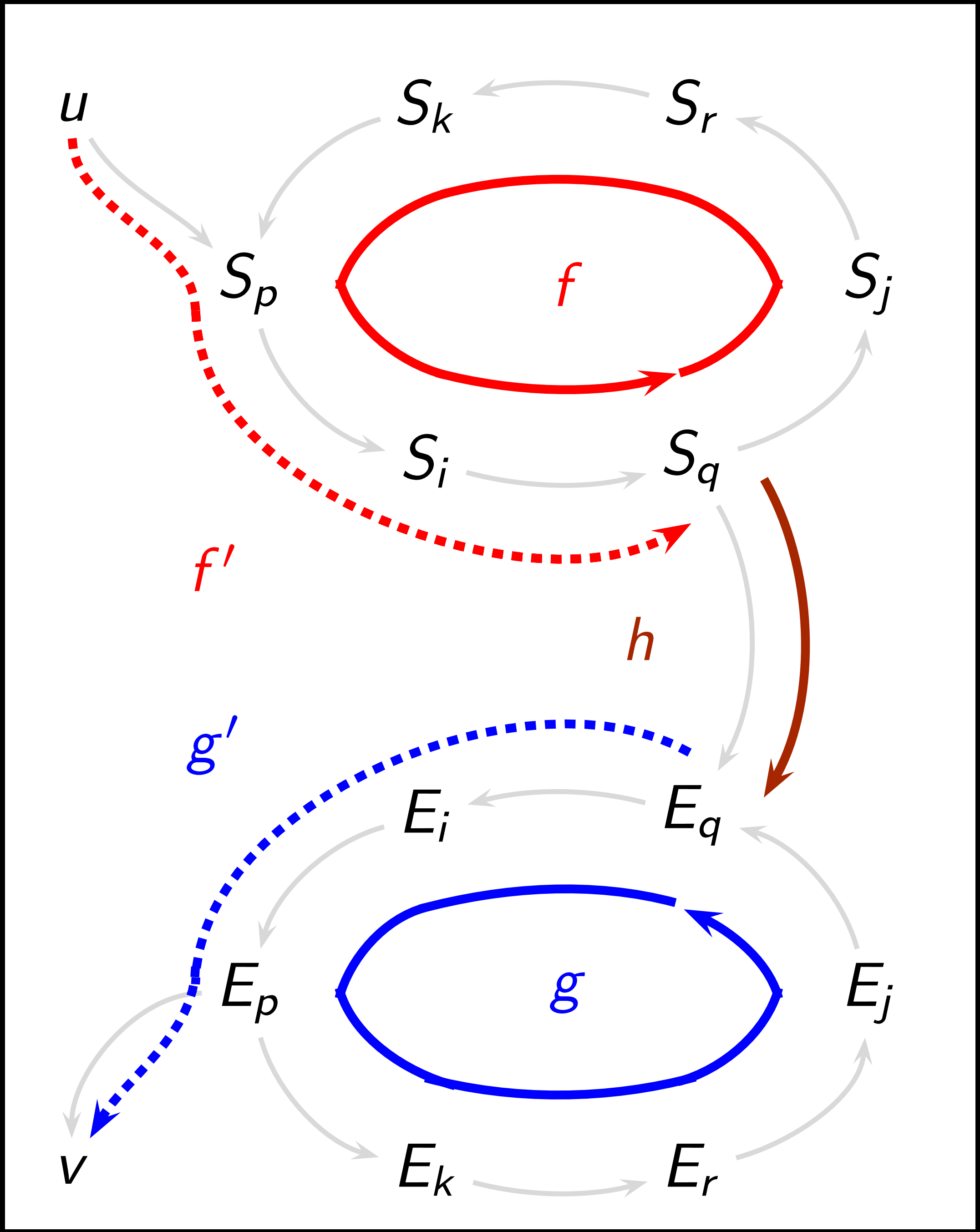
What About Recursion?

$$\begin{tikzcd}
u \arrow[dr] & & S_k \arrow[dl, bend right] & S_r \arrow[l, bend right] & \\
& S_p \arrow[dr, bend right] & & & S_j \arrow[ul, bend right] \\
& & S_i \arrow[r, bend right] & S_q \arrow[ur, bend right] \arrow[d, bend left] & \\
& & E_i \arrow[dl, bend right] & E_q \arrow[l, bend right] & \\
& E_p \arrow[dl, bend right] \arrow[dr, bend right] & & & E_j \arrow[ul, bend right] \\
v & & E_k \arrow[r, bend right] & E_r \arrow[ur, bend right] &
\end{tikzcd}$$

calls

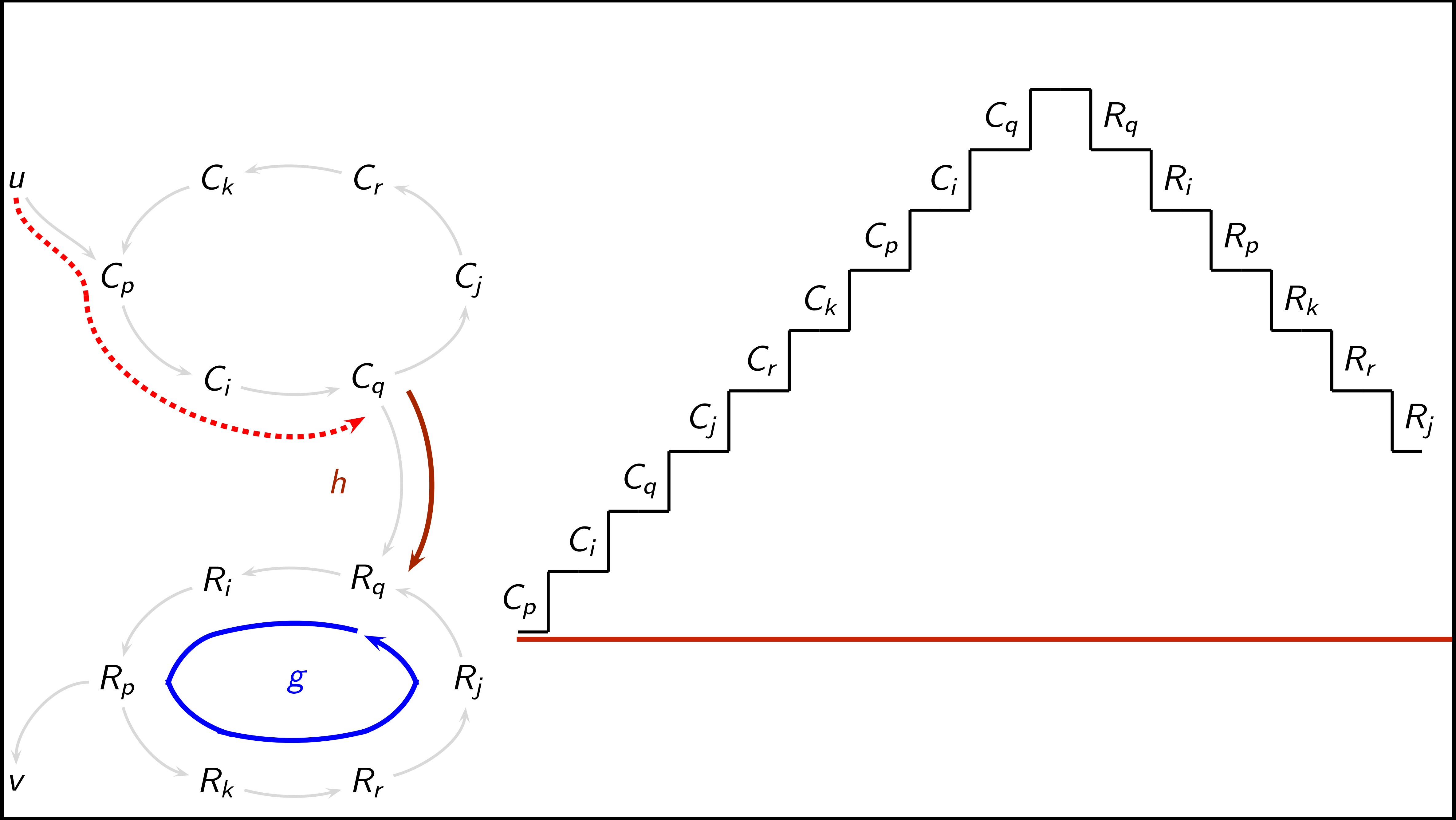
returns

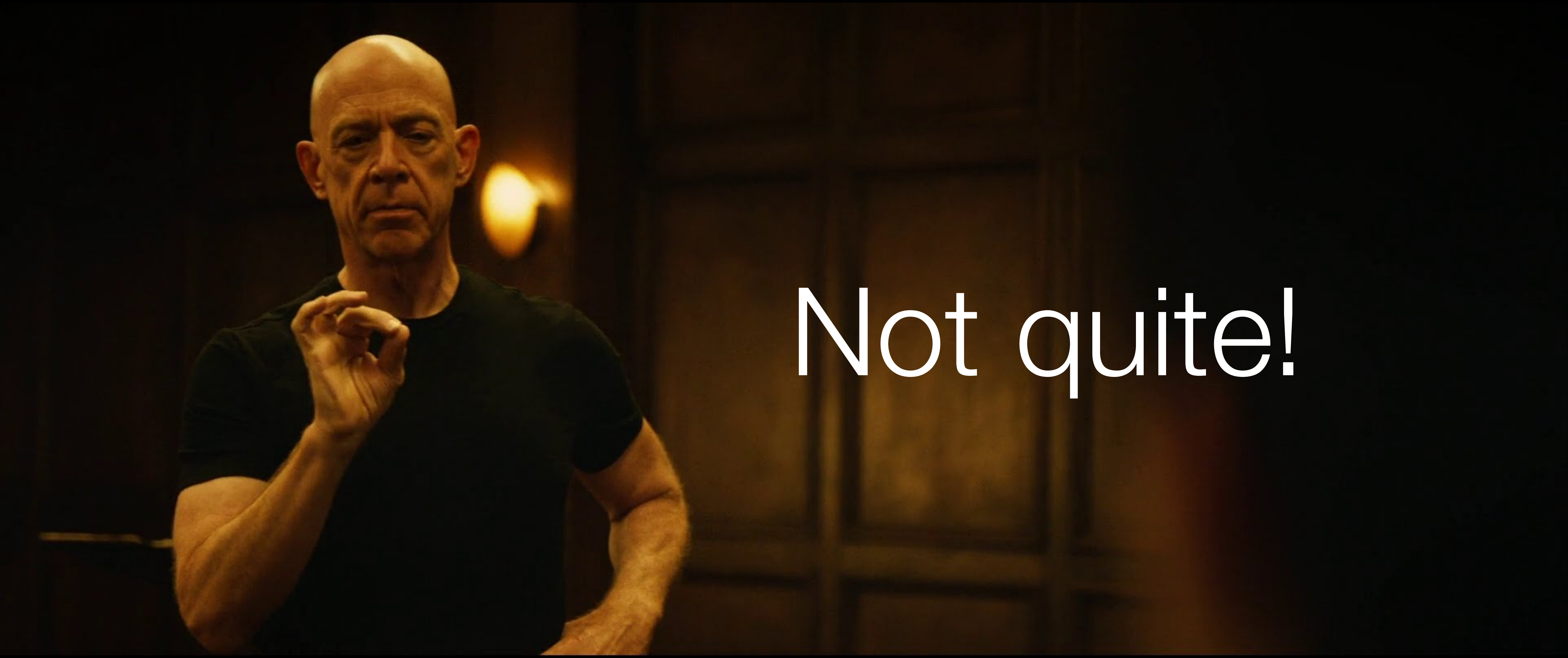
What About Recursion?



- For a path from u to v, g must be applied exactly the same number of times as f.
- For a prefix of the above path, g can be applied only at most as many times as f.

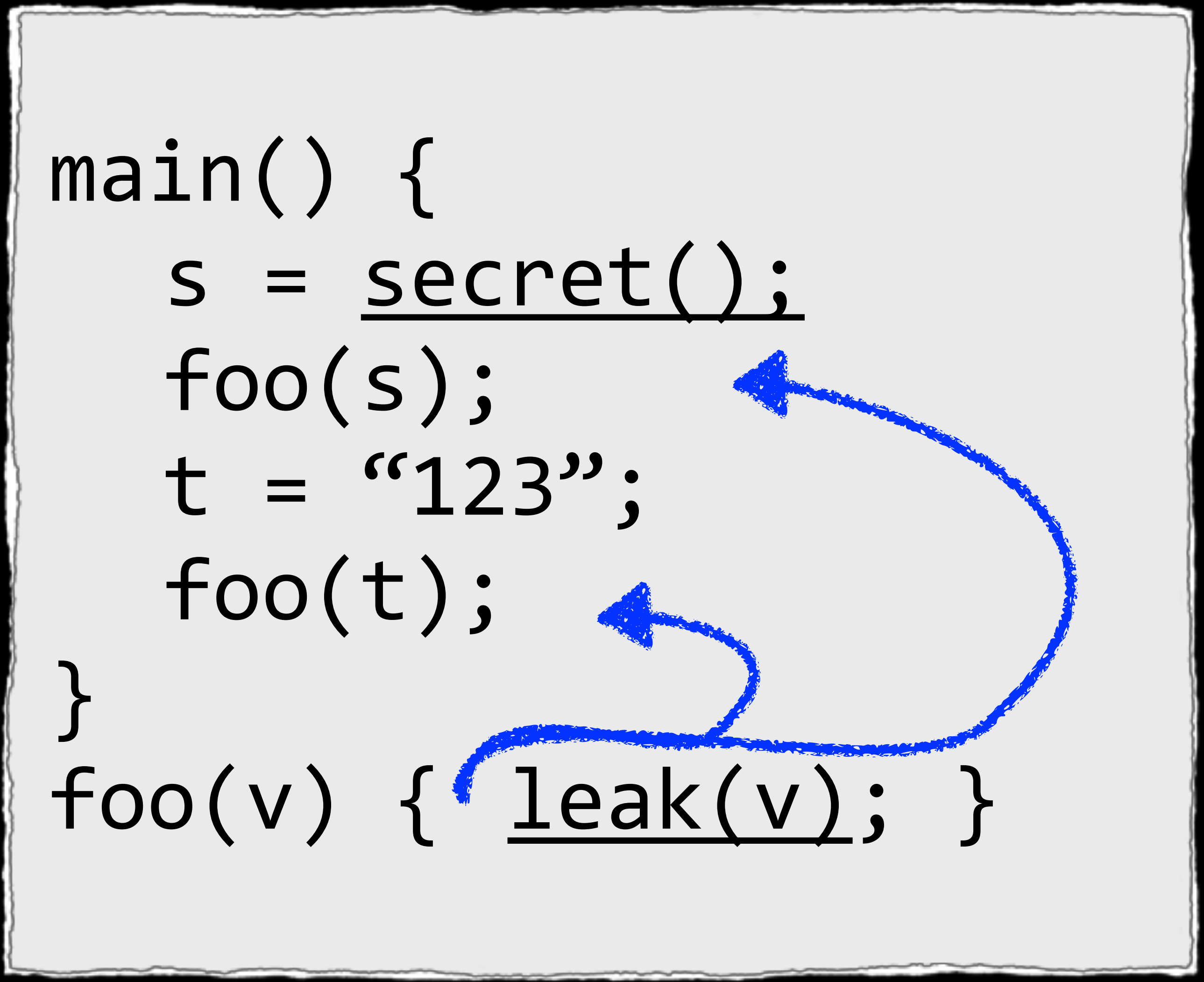
Staircase of Calls and Returns





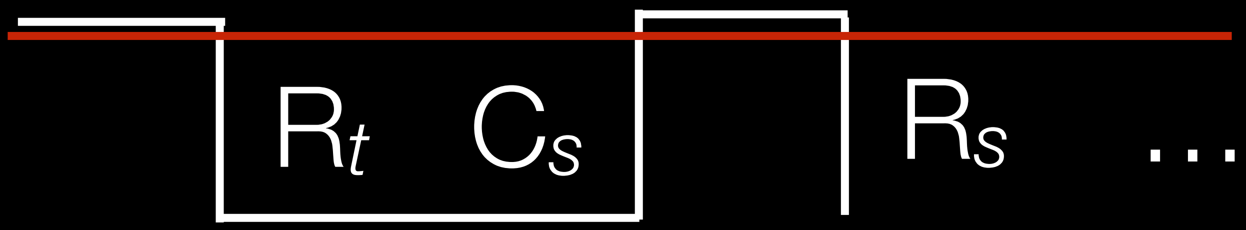
Not quite!

Demand-Driven Analysis



```
main() {
   s = secret();
   foo(s);
   t = "123";
   foo(t);
}
foo(v) { leak(v); }
```

assume we search from `foo(v)` backwards to find possible inputs



here: "unbalanced return" without a call must return to all possible callers

Solution =>
Context-Sensitive Analysis

Context-Sensitive Analysis

- Analyze the same method, depending on the context of the current *call* to that method

Context-Sensitive Analysis

- Considerations:
  - How to distinguish different contexts?
  - Which contexts can be merged?

Types of Context

- A call string that encodes the methods/call sites on the current call stack
- A value context that uses the input domain values as context
- An object context that uses the currently executing object as context
- and more…

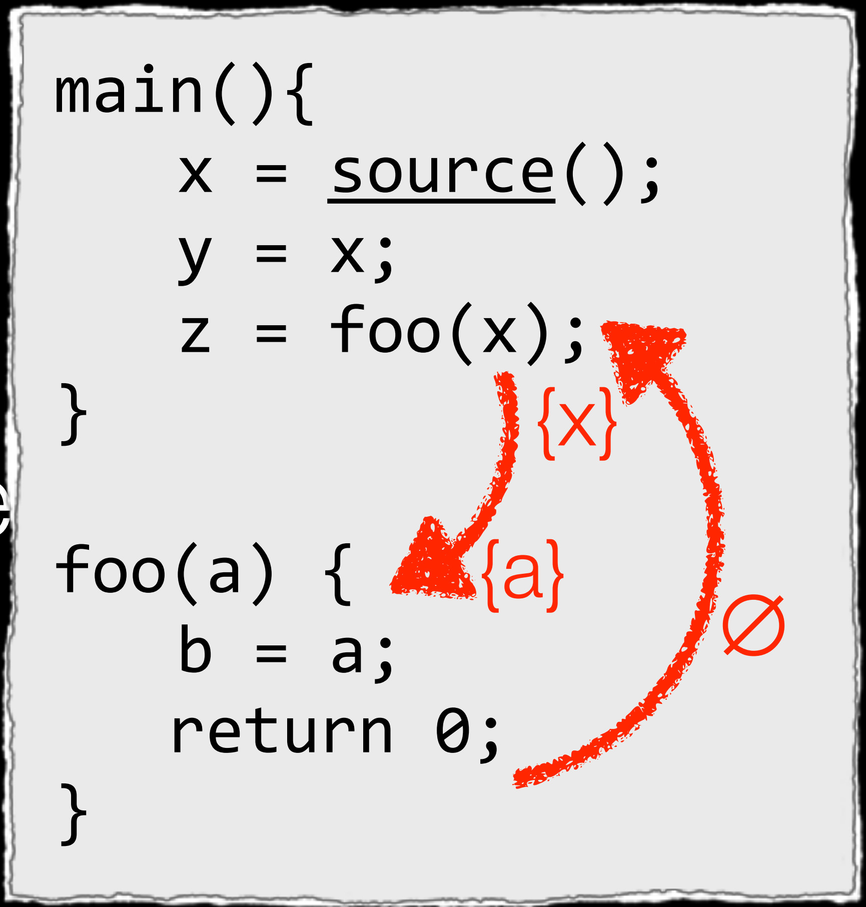
Important Language Features

Recursion

- Must bound computation and contexts
- Often uses flow-insensitive analysis to over-approximate

Parameters/Return Values

- Must map actuals to formals and vice versa
- Don't propagate too much info:
  - at a call: propagate only the facts relevant to that callee
  - at a return: propagate only the facts relevant to the caller
- Question: what to do with static fields?



```
main(){
    x = source();
    y = x;
    z = foo(x);
}

foo(a) {
    b = a;
    return 0;
}
```

Aliasing

aliases might be created by callers and callees

```
main(){
    a.f.g = source();
    foo(a,b);
    leak(b.f.g);
}

foo(x,y) {
    y.f=x.f;
}
```

Virtual Dispatch

- Multiple possible call targets per call site
- Consider them all!
  - “may” or “must” analysis?
  - similar to intra-procedural branches at if-then-else constructs (combine)

Threads

- Intra-procedural analyses are typically sound despite multi-threaded execution
- Inter-procedural analyses are typically *un*sound if flow-sensitive!
- Flow-insensitive analyses not impacted by multi-threading
- Effective modelling of synchronization constructs is a big open research problem!

Library Dependencies

- Typically analyze an application with its dependencies
- But what about native code?
- Often need to resort to hand-crafted summaries
- Possible way out: summarization (e.g., Averroes)

Recap

- Context-sensitivity analyzes a method multiple times, once per context
- Challenges: Recursion, parameters, aliasing, virtual dispatch, threads, libraries

Next

- IFDS